Structural design patterns - Adapter and Facade

Key terms

Structural Patterns

> Structural patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

> Structural patterns are concerned with how classes and objects are composed to form larger structures.

Adapter

> The adapter pattern is a software design pattern (also known as wrapper, an alternative naming shared with the decorator pattern) that allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

Facade

- A structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

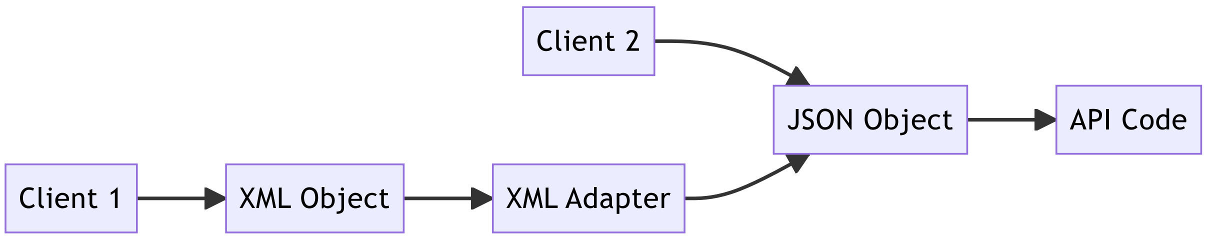
Adapter

> The adapter pattern is a structural pattern that allows objects with incompatible interfaces to collaborate.

We come across adapters in our day to day life. For example, we have a 3 pin plug and we want to use it in a 2 pin socket. We can use an adapter to convert the 3 pin plug to a 2 pin plug.

So, we use an adapter to allow two incompatible interfaces to work together. Similarly, in software development, we have two incompatible interfaces and we want to use them together. We can use an adapter to convert one interface to another. For instance, we have an API that returns a list of users. Now the request to this API requires a JSON object. Some clients instead of sending a JSON object, want to send an XML object.

Should we change the API to accept an XML object? Should we create a new API that accepts an XML object? No, that would be redundant. This is where the adapter pattern comes into play. We can create an adapter that converts the XML object to a JSON object and then use the existing API.



You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn’t even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here’s how it works:

- The adapter gets an interface, compatible with one of the existing objects.
- Using this interface, the existing object can safely call the adapter’s methods.
- Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Problem

Let us take the example of payment processing.
As a part of our application we want to integrate with different payment gateways. We first use the Stripe payment gateway. The stripe team provides us with a library that we can use to integrate with their payment gateway.

```
class StripeApi:
    def create_payment(self) -> Payment:
        # Create payment

    def check_status(self, payment_id) -> PaymentStatus:
        # Check payment status
```

We use the Stripe API to create a payment and check the status of the payment.

```
def process_payment():
    stripe_api = StripeApi()
    payment: Payment = stripe_api.create_payment()
    status: PaymentStatus = stripe_api.check_status(payment.id)
```

Now we want to integrate with another payment gateway. We use the PayPal payment gateway. The PayPal team provides us with a library that we can use to integrate with their payment gateway.

```
class PayPalApi:
    def make_payment(self) -> Payment:
        # Create payment

    def get_status(self, payment_id) -> PaymentStatus:
        # Check payment status
```

As you can see, the Stripe API and the PayPal API have different method names. The Stripe API uses `create_payment` and `check_status` while the PayPal API uses `make_payment` and `get_status`.
Should we change where we use the Stripe API to use the PayPal API? No, that would be redundant. That would require us to change the code in multiple places. Apart from the additional work, it would also increase the chances of introducing bugs. Also, when we want to switch back to the Stripe API, we would have to change the code again. Hence, our code is also violating SRP and OCP. We are also using concrete classes instead of interfaces. This makes our code tightly coupled.

Implementation

1. `Incompatible classes` - You should have two classes that have incompatible interfaces. For example, the Stripe API and the PayPal API.

```
class StripeApi:
    def create_payment(self) -> Payment:
        # Create payment

    def check_status(self, payment_id) -> PaymentStatus:
        # Check payment status

class PayPalApi:
    def make_payment(self) -> Payment:
        # Create payment

    def get_status(self, payment_id) -> PaymentStatus:
        # Check payment status
```

2. `Adapter interface` - Create an interface for the adapter that will be used to convert the incompatible interfaces.

```
from abc import ABC, abstractmethod

class PaymentProvider(ABC):
    @abstractmethod
    def make_payment(self) -> Payment:
        pass

    @abstractmethod
    def get_status(self, payment_id) -> PaymentStatus:
        pass
```

3. `Concrete adapter classes` - Create a class that implements the target interface. This is the class that the client code expects to work with. The adapter will convert the interface of the existing class to this interface.

```
class StripeAdapter(PaymentProvider):
    def make_payment(self) -> Payment:
        # Create payment

    def get_status(self, payment_id) -> PaymentStatus:
        # Check payment status

class PayPalAdapter(PaymentProvider):

    def make_payment(self) -> Payment:
        # Create payment

    def get_status(self, payment_id) -> PaymentStatus:
        # Check payment status
```

4. `Transform request and delegate to original class` - In the adapter class, transform the request to the format that the original class expects. Then, call the original class to perform the operation.

```
class StripeAdapter(PaymentProvider):
    def __init__(self):
        self.stripe_api = StripeApi()

    def make_payment(self) -> Payment:
        return self.stripe_api.create_payment()

    def get_status(self, payment_id) -> PaymentStatus:
        status = self.stripe_api.check_status(payment_id)
        return convert_status(status)
```

5. `Client code` - The client code expects to work with the target interface. The client code doesn't know that the adapter is converting the interface of the original class.

```
class PaymentProcessor:
    def __init__(self, payment_provider: PaymentProvider):
        self.payment_provider = payment_provider

    def process_payment(self):
        self.payment_provider.make_payment()
        status: PaymentStatus = self.payment_provider.get_status("paymentI
```

Advantages

- You can use adapters to reuse existing classes with incompatible interfaces.
- You can even modify the request and response of the original classes.

- Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program.
- Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the target interface.

Facade

> Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Facade means “face” in French. It is a front-facing building that is the main entrance to a building. The facade is the first thing that a visitor sees when they enter a building. The facade hides the complexity of the building from the visitor. The facade provides a simple interface to the building. The facade is a single point of entry to the building.

Problem

Let us take the example of an e-commerce application. The application has a lot of functionality. It has a product catalog, a shopping cart, a payment system, a shipping system, etc. The application has a lot of classes and a lot of dependencies between them. The application is complex and it is hard to understand how all the classes work together. When you make an order, you have to do the following:

- Call payment gateway to charge the credit card.
- Update the inventory.
- Email the customer.
- Add the order to the shipping queue.
- Update analytics.

The above steps are not trivial. The application has a lot of classes and a lot of dependencies between them. The application is complex and it is hard to understand how all the classes work together. The application is also hard to maintain. If you want to change the way the application sends emails, you will have to change the code in multiple places. If you want to add a new feature, you will have to change the code in multiple places. Imagine how the class looks:

```python
class Order:
    def __init__(self):
        self.payment_gateway = PaymentGateway()
        self.inventory = Inventory()
        self.email_service = EmailService()
        self.shipping_service = ShippingService()
        self.analytics_service = AnalyticsService()

    def checkout(self):
        self.payment_gateway.charge()
        self.inventory.update()
        self.email_service.send()
        self.shipping_service.add()
        self.analytics_service.update()
```

Here we have a lot of dependencies, some of which might be external vendors. The business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain. The Order class is hard to test. You will have to mock all the dependencies. The Order class is also hard to reuse. If you want to reuse the Order class in another application, you will have to change the code. Every time one of the logic changes, you will have to change the code in multiple places and hence violating SOLID principles.

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

Solution

The Facade pattern suggests that you wrap a complex subsystem with a simpler interface. The Facade pattern provides a higher-level interface that makes the subsystem easier to use. The Facade pattern is implemented by simply creating a new class that encapsulates the complex logic of the existing classes. For our example above, we will move the complex logic to a new class called OrderProcessor.

```
class OrderProcessor:
    def __init__(self):
        self.payment_gateway = PaymentGateway()
        self.inventory = Inventory()
        self.email_service = EmailService()
        self.shipping_service = ShippingService()
        self.analytics_service = AnalyticsService()

    def process(self):
        self.payment_gateway.charge()
        self.inventory.update()
        self.email_service.send()
        self.shipping_service.add()
        self.analytics_service.update()
```

Now we can use the OrderProcessor class in our Order class and delegate the complex logic to the OrderProcessor class.

```
class Order:
    def __init__(self, order_processor: OrderProcessor):
        self.order_processor = order_processor

    def checkout(self):
        self.order_processor.process()
```

The Order class is now much simpler. It has a single responsibility of creating an order. The Order class is also easier to test. You can mock the OrderProcessor class. The Order class is also easier to reuse. You can reuse the Order class in another application without changing the code.